Live Data Ingest Programmer's Guide

Integrated Test & Operations System
5 October 2006

Introduction

Applications in ITOS often need asynchronous notification of new telemetry data point values. The database interface (DBIF) API within the ITOS DBIF library, libdbif, provides a low-level mechanism applications use to arrange for this notification. The mechanism involves opening a DBIF "channel" (a FIFO; a.k.a. a named pipe) with dbTmOpenChannel(), and setting a "tag" on each mnemonic of interest with dbTmTagSet(). The DBIF then sends through the channel messages identifying a mnemonic and containing it's new value, which the application can read with dbTmReadChannel().

The Live Data Ingest (LDI) package provides a high-level, event-driven, and more object-oriented interface on top of the DBIF tag-and-channel mechanism for X Window System applications.[1]

The LDI functions according instructions contained in structures called `HandlingSpecs`, detailed below, and there are only three function calls to the LDI: The first, `LiveDataIngest create_live_data_ingest()`, creates the `LiveDataIngest` object instance. The second, `LiveDataIngest augment_ldi()`, can be used to add additional handling specs to a `LiveDataIngest` object, and the third, `int ldi_remove_hs()` can be used to remove handling specs from a `LiveDataIngest` instance..

The `HandlingSpecs` contains a list of telemetry mnemonics, specifies tagging and triggering options and provides hooks for two callback function which the application may provide. One callback is called each time a value message is received; the other is called when the trigger criteria given by the `HandlingSpecs` is met.

[1] Note that applications do not need to present a GUI to be X applications. The ITOS configuration monitor, eqn_cfgmon, is an X program with no GUI – it simply uses the programming facilities provided by X to make life easier for the programmer, and to give it access to the LDI!

1 DBIF Details

To better understand what the LDI does and how it operates, let's look at the relevant parts of the underlying DBIF layer and the database itself.

The ITOS operational database (in a part called the current value table or CVT) stores the value and some metadata for each telemetry mnemonic (that is, "measurand", a data point that lives in the ITOS operational database). Each mnemonic has a fixed value type, which can be signed or unsigned integers, floating-point, string, time, or date.

2 HandlingSpecs

The handling specs structure contains the following elements:

`TmnemGroup group;`
A group of mnemonics to tag.

`TagType tagtype;`
The type of tag to set. If 0, trigger by timed interval.

`Trigger trigger;`
The triggering criteria.

`Partial partial;`
How to handle missing values.

`int ques_qual_ok;`
If true, it's OK to use data with the QUALITY flag set.

`void (*trigger_callback)(PerSpecData *);`
The function to call when trigger criteria are met.

`void *trigger_data;`
Data passed as the third arugment to the trigger callback function.

`void (*value_callback)(INDEX, TmValue *, void *);`
The function to call when a new value has been received.

`void *value_data;`
Data passed as the third argument to the value callback.

`SpecHandle handle;`
An identifier for this HandlingSpec set by augment_ldi().

Details follow.

2.1 group

This is the set of mnemonics in which we're interested. The `TmnemGroup` is created from a `Null_Tmnem`-terminated list of `Tmnems` using the DBIF call `dbTmCreateGroup()`.

2.2 tagtype

The tag type concept is taken directly from the DBIF tag scheme. The idea is: In tagging a mnemonic, we can specify by the `TagType` the circumstances under which the DBIF should send us a value message.

The most useful distinction is: Send me a copy of the value whenever it is set; versus, send me a copy of the value only when it is **changed** from it's previous value.

From '`dbif.h`':

```
typedef int TagType; /* allowable tag types     */
/* (was an enum, now masks)     */
#define TmValChanged 0x1 /* send value when is changed     */
#define TmValSame 0x2 /* send value when is not changed   */
#define TmValSet 0x3 /* send value when is set     */
#define TmFlagChanged 0x4 /* send value when flags changed    */
#define TmFlagSame 0x8 /* send value when flags not changed*/
#define TmFlagSet 0xc /* send value when flags set     */
#define TmPktTag 0x10 /* this is a packet tag     */
```

`TmValChanged`
: Send a value message only when the new value is different from the old value.

`TmValSame`
: Send a value message only when the new value is the same as the old value. I can't imagine why anybody would want to do that; it's just here for completeness.

`TmValSet`
: Send a value message anytime the value is set, regardless of whether or not it matches the previous value.

`TmFlagChanged`
: Send a value message only when the new flags are different from the old flags.

`TmFlagSame`
: Send a value message only when the new flags are the same as the old flags. Again, I can't imagine why anybody would want to do that.

`TmFlagSet`
: Send a value message anytime the flags are set, regardless of whether or not they match the previous flags.

2.3 trigger

From 'live_data_ingest.h':

```
typedef int TriggerType; /* trigger types     */

#define TT_NULL 0 /* no trigger     */
#define TT_TIME 0x1 /* trigger on timed interval     */
#define TT_NORM 0x2 /* normal triggering     */
#define TT_EOM  0x4 /* trigger on DBIF MSG_END_MARKER   */
#define TT_MNEM 0x8 /* trigger on receipt of certain
   mnemonics     */
#define TT_PKT 0x10 /* trigger on receipt of certain
   packets     */
```

2.4 partial

From 'live_data_ingest.h':

typedef enum PL_NULL, /* unset */ PL_OK, /* leave missing values out */ PL_FILL, /* fill missing values from CVT */ PL_DISCARD /* suppress trigger if vals missing */ Partial;

2.5 ques_qual_ok

If this element is zero, values flagged as having questionable data quality will *not* generate value callbacks and be used to determine if the triggering criteria have been met.

2.6 trigger_callback

This element is a pointer to a function returning void (no return value) and taking a single argument which is a pointer to a `PerSpecData` structure. This function is called when the triggering criteria are met.

The PerSpecData structure encapsulates the data associated with the particular HandlingSpec for which the *trigger_callback* is being called. For most purposes, it can be treated as an opaque object from which you can access a list of `PerMnemData` structures, each of which carries data related to a single mnemonic called for in the `HandlingSpec`.

Use the macro `FIRST_PMD()` to get the first PerMnemData structure in the list, and `NEXT_PMD()` to get subsequent entries in the list. `NEXT_PMD()` will return `NULL` when it reaches the end of the list.

The PerMnemData structure contains the following elements:

`INDEX idx;`
: index in IngestData.mnem_list

`Tmnem mne;`
: mnemonic's DBIF handle

`TmInfo info;`
: mnemonic's DBIF information

`TmValue tmval;`
: mnemonic's DBIF value

`TmTag tag;`
: mnemonic's DBIF tag, if needed

`PerSpecDataLink *hs_list;`
: list of handling specs which reference this mnemonic

For most programmers, only the `tmval` field containing the mnemonic's value will be of use.

For details on the `PerSpecData` structure, See Section 2.6.1 [per_spec_data], page 6.

2.6.1 PerSpecData Strucutre

From `live_data_ingest.h`:

```
/* Data for each handling spec, including the list of mnemonics
   referenced by the spec.  A pointer to one of these is passed to the
   trigger_callback function.  The _PMD macros following may be used
   to iterate through each PerMnemData in the "mnem_list".      */

struct per_spec_data

    HandlingSpecs hs; /* data handling specifications     */
    PerMnemDataLink *mnem_list; /* list of mnemonic data for spec   */
    IngestData *ldi; /* pointer back to containing LDI   */
    PerMnemDataLink *pmdl; /* supports FIRST, NEXT macros below*/
;

#define ldi_psd_ldi(psd) ((psd)->ldi)
#define FIRST_PMD(psd) ((psd)->pmdl = (psd)->mnem_list, (psd)->pmdl->pmd)
#define NEXT_PMD(psd)   ((psd)->pmdl = (psd)->pmdl->next, (psd)->pmdl->pmd)

/* A linked list of pointers to PerMnemData.

   There is one of these lists in each PerSpecData structure giving
   the list of mnemonics referenced by the spec.  For each element in
   such a list, there is a one-to-one correspondence to a
   PerSpecDataLink in pmd->hs_list.      */

struct per_mnem_data_link

    PerMnemData *pmd; /* pointer to associated mnem data  */
    PerMnemDataLink *next; /* link to next mnemonic data ptr   */
    PerSpecMnemData *psmd; /* pointer to per-spec mnem data    */
;
```

2.7 trigger_data

The `trigger_data` element is a pointer to data which the creator of the `HandlingSpec` wants to make available to the `value_callback` function. Currently it has to be accessed through the callback's `PerSpecData *` argument, `->hs.trigger_data`.

2.8 value_callback

This element is a pointer to a function returning void (no return value) and taking a three arguments with the following types:

`INDEX` the array index for the mnemonic associated with the value in the `TmnemGroup` given to the handling spec.

`TmValue *` a pointer to the mnemonic's value.

`void *` The `value_data` included in the `HandlingSpec`.

2.9 value_data

This element is a pointer to data which the creator of the `HandlingSpec` wants passed to the `value_callback` function.

2.10 handle

This element is set by `augment_ldi()` and `create_live_data_ingest()` in the `HandlingSpecs` structure to which they are passed a pointer. It must be saved and passed to `ldi_remove_hs()` to remove this `HandlingSpecs` from the LDI.

3 create_live_data_ingest

Syntax

```
LiveDataIngest create_live_data_ingest(XtAppContext context,
    TmChan channel, HandlingSpecs *hs, void (*errcb)(int, char *, ...))
```

Discussion

This function creates an instance of the Live Data Ingest (LDI). The LDI is intended to work within standard Xt-based applications, so the first argument, *context*, is the Xt application context.

The *channel* argument is a DBIF channel created by `dbTmOpenChannel()` or `NULL`. If it's not `NULL`, this is the DBIF channel which should be used to receive telemetry values if the `HandlingSpecs trigger_type` element is not `TT_TIME`. If it is `NULL` and a channel is needed, the LDI will open one automatically. This argument usually can be set to `NULL`, but is provided for cases where the application already has a channel open.

The *hs* argument is a pointer to the first (or only) set of handling specs for this LDI. This argument also may be `NULL`, in which case a the first spec, like any others, must be added with `augment_ldi`.

The *errorb* argument is for passing an error callback function to the LDI. If non-`NULL`, this should be a pointer to a function with the signature:

```
void errfn(int status, char *format, ...)
```

If *status* is zero, the message is informational, not really an error; if it is less than zero, it is an error; and if it is greater than zero, it is a warning. The *format* is an `EvtMsg`-style format string.

The return value is the LDI instance which must be passed to `augment_ldi()` and `ldi_remove_hs()`.

4 augment_ldi

Syntax

```
LiveDataIngest augment_ldi(LiveDataIngest ldi, HandlingSpecs *hs)
```

Discussion

This function adds the `HandlingSpecs` *hs* to the existing `LiveDataIngest` instance *ldi*. It returns it's first argument or `NULL` on errors.

5 ldi_remove_hs

Syntax

```
int ldi_remove_hs(LiveDataIngest ldi, SpecHandle handle)
```

This function removes the `HandlingSpecs` given by *handle* from the `LiveDataIngest` instance *ldi*. The *handle* is a field in the `HandlingSpecs` set by `augment_ldi()` and `create_live_data_ingest()` in the `HandlingSpecs` structure to which they are passed a pointer.

The function returns a non-zero value on errors.

6 Internal Details

The following is an attempt to illustrate the interlocking nature of the two lists in the LDI: the handling specification list, hs_list, and the mnemonic list, mnem_list.

```
PSDL* hs_list
-------------
    PSD*  psd <------------------+
    ---------                    |
        HS                       |
        LDI                      |
        PMDL* mnem_list          |
        ---------------          |
            PMD*  pmd  ------+   |
            PSMD* psmd ------|---|----+
            PMDL* next --+   |   |    |
            ----------   |   |   |    |
                             |   |    V
    PSMD* psmd               |   |
    ----------               |   |    PSMD
        unused               |   |    ----
                             |   |    INDEX spec_idx
    PSDL* next --+           |   |    int   recd
    ----------   |           |   |
                 |           |   |    ^
           ...<--+           |   |    |
                             |   |    |
                             |   |    |
PMD*  mnem_list[] <----------+   |    |
-----------------                |    |
    INDEX idx                    |    |
    Tmnem mne                    |    |
    TmInfo info                  |    |
    TmValue tmval                |    |
    TmTag tag                    |    |
    PSDL* hs_list                |    |
    -------------                |    |
        PSD*  psd ---------------+    |
        PSMD* psmd -------------------+
        PSDL* next --+
        ----------   |
                     |
               ...<--+

    ...
```

As shown, each element in the hs_list basically just contains a PerSpecData structure. This contains the HandlingSpec and a list of mnemonics reference by the spec. Each element in the mnemonic list, `mnem_list`, contains a pointer to an entry in the LDI's mnemonic_list, and a pointer to a `PerSpecMnemData` structure. This latter structure is used to keep track of what mnemonics have been received for this spec, and how each mnemonic is identified relative to the spec.

Each element in the LDI's mnemonic list, `mnem_list`, contains all data pertinent to a mnemonic and a list of handling specifications which refer to the mnemonic. The mnemonic's handling spec list has the same structure as the LDI's list of all handling specs.

There are two views into the mnemonic data: the data ingest view and the triggering view. In the former, per-mnemonic data is layed out in a list which can be indexed directly by a dbif tag marker. This is exactly what happens when tagged data arrives. In the triggering view, each handling spec has to be examined to see if it's triggering criterion has been met. For this purpose, per-mnemonic data is organized into a linked list with each item containing data for one handling spec.

struct ldi_internal_data

int num_hs
number of handling specs

PerSpecDataLink *hs_list
linked list of handling spec data

int num_mnems
count of unique mnemonics in ldi

PerMnemData **mnem_list
array of per-mnemonic data

TmChan chan
DBIF channel for data input

Timer *timers
Xt timer ID for interval trigger

XtAppContext xt_app_context
Xt application context

void (*error_cb)(int, char *, ...)
callback function for errors